Basics of programming language: An introduction using ARLA

Antonio Calcagnì
DPSS - University of Padova

Contents

In this short course, we will review the basics of programming language using ARLA ("Another R LAnguage"), a like-R pseudo-code based on structured programming paradigm created to introduce scientific computing within the R environment (`https://cran.r-project.org/`). These notes are intended to provide some guidance for those who wish to learn how to program using an imperative and structured programming paradigm. They are written at an introductory level needed to solve most of the basic problems encountered in scientific computing. Note that the current version does not cover all the topics typically encountered in a programming course like programming with strings and characters. These parts will be treated in a further version.

Contact: `antonio.calcagni@unipd.it`
Version: 1.0 (October, 2019)

1 Type of data

A first important definition is that of variable and costant. A **variable** is a physical location (part of a computer memory) where information are stored. They are typically described by a symbolic *name* (e.g., "X"), a *value* (e.g., 105), which is contained in the memory location, and *type* (e.g., numeric, string), denoting the set of admitted operations which can be performed on variables (e.g., sum, concatenation). Variables can be thought as boxes that can be updated and modified in a number of ways during the execution of the program[1]. By contrast, a **costant** is a kind of storage location which cannot be neither updated nor modified during the execution of the program. Likewise for variables, constants can be thought as closed boxes. Variables and constants also differ in the way they access the computer memory: whilst variables require updating the storage location over and over during the execution of the program, constants do not, as they are assigned before the execution is started.

Data can be also classified in terms of their use. Hence, we have **input** data (external information required by the program to execute the commands), **output** data (information provided by the program as a result of its processing), and **working** data (intermediate information required to link input to output).

Example 1: Area of a square

To compute the area of a square we apply the simple formula $A = \frac{1}{2}d^2$, with d being the diagonal of the square. The input of the program is therefore d and the output A. The value $\frac{1}{2}$ is a constant. If we split the computation into two parts, for instance (a) $b = d^2$ and (b) $A = \frac{1}{2}b$, then b can be considered a working variable.

Symbolic names of variables usually do not contain any special characters like ?, !, %. Names should be kept as simple as possible and composite names can be formed by using the symbol _, like `var_a`.

Types can be classified into *numeric*, *integer*, and *string*. Numeric variables are data expressed by numbers, like integers $(-1, 0, 1, \ldots)$ and reals $(2.34, 0.91, -9.1987)$. Strings are expressed by letters of the alphabet, e.g. "hello world". The string "hello_world 1" is an alphanumeric variable, which is formed by characters and numbers as well (in this case 1 is treated as string).

Boolean data are of the type TRUE/FALSE (1/0) and are of particular interest when the program involves logical operators like AND, OR, NOT.

Data can also be classified into simple and structured. *Simple* data are information expressed by means of un-indexed units whereas *structured* data are aggregation of that are simple units usually indexed by natural numbers. Thus, for instance, the area of a square (see Example 1) is an example of simple data whereas **arrays** like vectors or matrices, or rather **lists** or **dictionaries** are example of structured/aggregated data.

2 Order of mathematical operations

When executing a program containing mathematical expression, as for the case described in Example 1, programming languages follow the order:

- Exponents
- Multiplication and division
- Addition and subtraction

This helps the interpreter to reduce the eventual ambiguity present in the program. For instance, the operation $x = \frac{y^2}{0.5} + 3.4z$ is executed in the following way: (i) y^2, (ii) $\frac{y^2}{0.5}$ and $3.4z$, (iii) all

[1] A **program** is a set of finite and ordered instructions needed to perform a work (output) given some initial information (input).

together $\frac{y^2}{0.5} + 3.4z$ by using the previous results. To force the execution of a particular block, we need to parenthesize mathematical expressions by means of ( ). For example, $x = (\frac{y^2}{0.5} + 3.4)z$ computes the expression inside the parentheses first even though multiplication has priority on addition. Note that *Square* and *curly* brackets are instead used to access elements of arrays and other structures like lists or dictionaries.

3 Declaring variables

Before using variables or arrays, there is need to *declare* them. Declaration allows programs to handle with variables w.r.t. admitted operations (e.g., mathematical operations, string operations). Many modern programming languages like Python and R admit dynamic variable declaration where the interpreter automatically detect the type of variables while it reads their content. In this sense, there is no need to declare variables before their use. However, didactically speaking, declaring variables is still useful, particularly because it can help programmers to think about the type of information the program needs to work properly. In ARLA, variables are declared by typing the command:

`(variable name)` := (TYPE)

Examples of variables declaration include:

`x` := BOOLEAN
`y` := NUMERIC
`z` := STRING

Programs treat variables as they are declared. Therefore, it is not possible to declare a variable as string and to use it to store integers.

4 Statements

Statements are *commands* through which a program executes some operations. The first and most basic statement is **assignment**:

`a` = 2.3
`a_txt` = "hello_world"

It simply assigns a new value to simple or structured variables. This command can also be used with expressions:

`x` = 2.3
`y` = 10.1
`z` = `x` ·2.3 + 0.5·`y`

where, in the third case, the variable `z` contains the results of the expression involving constants (i.e., 2.3, 0.5) and previously defined variables (i.e., `x`, `y`). Different values can also be assigned to same variables:

`x` = 2.3
`x` = 3.4

with the consequence that previous content of `x` will be overwritten.

A second basic statement is **read**, which allows the program to read and use external information:

`x` = **read**(*Something...*)

The command asks the program to wait for a user input (e.g., by pressing buttons on a keyboard) before executing further operations. On the contrary, the statement **print** allows the computer program to print the values of variables on a external device (e.g., user's console):

```
x  = read(Something...)
print(x)
```

Example 2: Area of a square (algorithm)

```
d := Numeric
a := Numeric
Begin
   d  = read("Write the diagonal of your square")
   a  = d^2 · 0.5
   print("The area of the square is:", a)
End
```

Example 2 shows a simple program to compute the area of a square with ARLA.

5 Control structures

In structured programming, each program is written in blocks (subprograms), each of them requiring unique input and output. Blocks are organized together by means of sequences, selections, and iterations (loops). As stated by Bohm and Jacopini theorem, these structures represent the three control structures at the base of any algorithm.

5.1 Sequence

Sequence is the simplest structure allowing programs to execute operations or blocks sequentially "as they are" written. Example 3 shows the algorithm used to swap the values of two distinct variables `a` and `b`.

Example 3: Swap values of two variables

```
a := Numeric
b := Numeric
c := Numeric
Begin
   a  = read("Write a number")
   b  = read("Write another number")
   c  = a
   a  = b
   b  = c
   print(a,b)
End
```

To test the correct execution of this program, we can use the table *commands* × *variables* (see Table 1).

statement	a	b	c	output
read("Write a number")	2			
read("Write another number")		6		
c = a			2	
a = b	6			
b = c		2		
print(a,b)				6, 2

Table 1: Table commands × variables to test programs

5.2 Selection

Selection allows programs to make choices among alternatives. The simplest form of this structure is the **one-way** selection:

if(*condition*){*do something*}
else{*do something else*}

where the first block (*do something*) is executed if *condition* is met; otherwise, the second block of commands (*do something else*) is instead executed. Blocks of commands need to be inside curly brackets. Conditions must be written using logical operations like == (equal), > (greater than), < (less than), ! = (is not) and logical connections & (and), | (or). Some examples on how if-then-else can be used are shown below:

```
if(x>0.5){
a = x+1}
else{
a = x-1}
```

```
if(x>0.5 & y==(x/2)){
a = x+y}
else{
a = x-y}
```

```
if(x<0.5 & y! =(x/2)){
a = x+y}
else{
if(z<x+y){ a = x^2 / y}
}
else{a = 0}
```

The third example shows a nested if-else structure where, if the first condition is not met, then the program is asked to control another if-then condition before executing the last else form.

Example 4: Verify that a number N is positive or negative

```
N := Numeric
Begin
  N  = read("Write a number")
  if(N>0){
  print(N "is positive")}
  else{
  print(N "is negative")}
End
```

Example 5: Verify that a number N is positive, negative, or null

```
N := Numeric
Begin
  N  = read("Write a number")
  if(N>0){ print(N "is positive")}
  else{
  if(N<0){ print(N "is negative")}
  else{ print(N "is null")}
  }
End
```

When the program has to verify interval conditions such as:

$$2.3 < x \leq 9.8$$

the condition in the if-then command needs to be written in the logical form:

$$\texttt{x} > 2.3 \ \& \ \texttt{x} <= 9.8$$

An example of this form is shown in Example 6.

Example 6: Check if a number N is inside a given interval $[a, b]$

```
N := Numeric
a := Numeric
b := Numeric
Begin N  = read("Write a number")
  a  = read("Write the lower bound of the interval")
  b  = read("Write the upper bound of the interval")
  if(N >= a & N <= b){print(N "is in inside the interval")}
  else{print(N "is in outside the interval")}
End
```

The correctness of Example 6 can be verified by means of a commands × variables table.

5.3 Loops

Loops are used to automatize instructions that need to be repeated many times. A first form is **for-loop** where blocks of commands are iterated through an indexed set. In this case, the number of iterations are known prior the programs runs. This control structure is written by using the following sintax:

for(`i` **in** `1:n`){*do something*}

where `i` is the variable used to go through the index set `1:n`, which is in turn used to repeat n times the block *do something*. Alternatively, the structure can be written as follows:

`n` = 1:`n`
for(`i` **in** `n`){*do something*}

where the index set `n` is declared before running the loop.

Example 7: Summing N (input) numbers

```
x := Numeric
sum_var := Numeric
Begin
  sum_var = 0
  for(i in 1:N ){
  x = read("Write a number")
  sum_var  = sum_var + x
  }
  print("The final sum is" sum_var)
End
```

Example 7 shows a program that uses the variable `sum_var`, which progressively updates the final sum by the amount stored in `x` (the number given in input). Note that `sum_var` has been explicitly *initialized* before the loop (`sum_var` = 0). Table 2 shows a test for the sum of five numbers.

statement	i	x	sum_var	output
`sum_var` = 0			0	
read("Write a number")	1	2.5		
`sum_var` + `x`			2.5	
read("Write a number")	2	0.5		
`sum_var` + `x`			3	
read("Write a number")	3	-1.0		
`sum_var` + `x`			1.5	
read("Write a number")	4	9.3		
`sum_var` + `x`			10.8	
read("Write a number")	5	-0.8		
`sum_var` + `x`			10	
print("The final sum is" `sum_var`)				10.0

Table 2: Table commands × variables for Example 7

Another type of loop is **while** where a block of commands is executed until a specific condition is met. It uses the following syntax:

while(`condition`){*do something*}

Unlike for the case of **for-loop**, this structure asks program to run the block of commands with no knowledge of when execution must be stopped.

Example 8: Summing N (input) numbers – second version

```
x := Numeric
sum_var := Numeric
N := Numeric
count := Numeric

Begin
  N = read("How many numbers...?")
  sum_var = 0
  count = 0
  while(count < N ){
  count = count+1
  x = read("Insert a number")
  sum_var = sum_var + x
  }
  print("The final sum is" sum_var)
End
```

Example 8 shows a case using *counters*, i.e. a variable that counts the number of iterations the programs does to reach the stopping condition.

statement	N	count	x	sum_var	true/false	output
sum_var = 0				0		
count = 0		0				
read("How many numbers...?")	3					
count < N					V	
count = count+1		1				
read("Write a number")			3			
sum_var + x				3		
count < N					V	
count = count+1		2				
read("Write a number")			5			
sum_var + x				8		
count < N					V	
count = count+1		3				
read("Write a number")			-6			
sum_var + x				2		
count < N					F	
print("The final sum is" sum_var)						2

Table 3: Table commands × variables for Example 8

In some circumstances, it can be required that the *stopping condition* must be reached as a results of some instructions executed inside the loop. In this case, if the stopping condition is badly-written, the program will loop endlessly (*infinite loop* problem). Example 9 shows a typical case where the stopping condition is inside the block of commands. Example 10 describes a more complete program where different control structures are used together to compute the mean of a subset of positive numbers from N numbers.

Example 9: Summing N (input) numbers with conditioning

```
x := Numeric
sum_var := Numeric
N := Numeric
cond := Boolean
Begin
  N = read("How many numbers...?")
  sum_var = 0
  cond = 0
  while(cond != 1 ){
  x = read("Insert a number")
  sum_var = sum_var + x
  if(sum_var > N*2){ cond = 1}
  }
  print("The final sum is" sum_var)
End
```

statement	N	cond	x	sum_var	true/false	output
sum_var = 0				0		
cond = 0		0				
read("How many numbers...?")	10					
cond != 1					V	
read("Write a number")			3			
sum_var + x				3		
sum_var > N*2					F	
cond != 1					V	
read("Write a number")			6.5			
sum_var + x				9.5		
sum_var > N*2					F	
cond != 1					V	
read("Write a number")			18			
sum_var + x				27.5		
sum_var > N*2					V	
cond		1				
cond != 1					F	
print("The final sum is" sum_var)						27.5

Table 4: Table commands × variables for Example 9

Example 10: Mean of a subset of N (input) numbers

```
x := Numeric
sum_var := Numeric
N := Numeric
count := Numeric
count_pos := Numeric
mean := Numeric
Begin
  N = read("How many numbers...?")
  sum_var = 0
  count = 0
  count_pos = 0
  while(count < N ){
  count = count + 1
  x = read("Insert a number")
  if(x > 0){
  sum_var = sum_var + x
  count_pos = count_pos + 1 }
  }
  mean  = sum_var / count_pos
  print("The mean is" mean)
End
```

6 Subprograms and functions

Writing a program requires analysing and solving specific sub-problems by means of a good strategy. The **top-down** approach is a common strategy to solve problems by decompose them in many sub-problems each with a unique solution. For instance, writing a program to compute the arithmetic mean of N numbers can be done by implementing a first program (sub-routine), which computes the sum of N inputs, and a second program (main routine), which instead uses the output of the sub-routine to compute the arithmetic mean. The sub-routine is a complete and coherent sub-program that can be used regardless of the main routine (i.e., it is independent from the computations required by the main routine). A first and direct advantage of sub-routines is that they help programmers to debug their programs quickly. A subroutine can be created as follows:

```
name_function := function(input ){
do something
return(output)}
```

and it is called by the following syntax:

```
out  = name_function(x,y,...)
```

where `out` contains the results passed by **return(...)**. Example 11 shows an example of sub-routine in the simple program for the arithmetic mean of N numbers.

Example 11: Mean of N (input) numbers with sub-routine

```
x := Numeric
sum_var := Numeric
N := Numeric
count := Numeric
mean := Numeric
Begin
  N = read("How many numbers...?")
  sum_var = 0
  count = 0
  while(count < N ){
  count = count + 1
  x = read("Insert a number")
  sum_var  = compute_sum(sum_var,x)
  }
  mean  = sum_var / N
  print("The mean is" mean)
End

compute_sum := function(a,b ){
out := Numeric
out  = a+b
return(out)
}
```

Likewise for variables in main routines, arguments passed in the function as input are of the same type of those declared in the main routine (same declaration). For instance, in Example 11 `a` and `b` are declared as numeric before their use in the sub-routine. This constraints the type of arguments to be passed and check for the type of admitted operations in the sub-routine. Thus, if we pass the strings "hello" and "world" to `compute_sum(..)`, the interpreter of the program will raise an exception about the admitted operation declared in `compute_sum()` (indeed, no mathematical operation + is defined for strings).
There are functions that do not need to be declared before their use. They are built-in routines, called **primitive functions**, which execute low-level instructions. Examples of them include mathematical and set operators. Table 5 shows primitive functions available in ARLA.

`concatenate()`	Concatenate single elements like numbers or strings
`matrix(data,nrow,ncol,by=ROWS/COLS)`	Populate a `nrow` × `ncol` matrix
`length()`	Number of elements of a vector
`NROW()`	Number of rows of a matrix
`NCOL()`	Number of cols of a matrix
`abs()`	Absolute value
`round(x,digits=n)`	Rounding numbers to n digits
`cos()`, `sin()`, `tan()`	Some trigonometric functions
`log(x,base=n)`	Logarithm function
`exp()`	Exp function
`sqrt()`	Square root function

Table 5: Built-in functions in ARLA

7 Arrays

Arrays are a particular type of **structured data**, namely variables or constants that contain more information *of the same type* (e.g., numeric). Arrays are multidimensional structures where each dimension is indexed by a numeric variable called *index*.

7.1 Vectors

Arrays with just one dimension are called **vectors** and are written as follows:

```
(name) := NUMERIC[N]
(name) := BOOLEAN[N]
(name) := STRING[N]
```

where N is the length of the vector, i.e. the number of elements that it will contain. Note that N must be always greater than zero. The case $N = 1$ is a special vector with just one element (singleton). Vectors can be populated using a special for-loop (see Example 13) or via `concatenate`(), which is a primitive function used to put together different elements:

```
x := NUMERIC[5]
x[]  = concatenate(1,2,10,76,-2)
```

where the number of elements used in `concatenate`() must equal the length of `x[]`. Example 12-13 show typical examples of programs using vectors.

Example 12: Sorting two (input) numbers

```
x := NUMERIC[2]
y := NUMERIC[2]
BEGIN
   x[1]  = read("Write the second number")
   x[2]  = read("Write the second number")
   y[]  = invert(x[])
   print("The resulting vector is" y[])
END

invert := function(a[] ){
c := NUMERIC
if(a[1]>a[2]){
c  = a[2]
a[2]  = a[1]
a[1]  = c}
return(a[])
}
```

Example 13: Mean of N (input) numbers with vectors

```
N := NUMERIC
mean := NUMERIC
x := NUMERIC[N]
BEGIN
   for(i in 1:N ){
   x[i]  = read("Write a number")
   }
   mean = compute_sum(x[]) / N
   print("The mean is" mean)
END

compute_sum := function(a[] ){
sum := NUMERIC
N := NUMERIC
N = length(a[])
sum = 0
for(i in 1:N ){ sum  = sum + x[i]}
return(sum)
}
```

In Example 13 the for-loop inside the main routine populate the vector `x[]` with a set of inputs provided by the user (it constitutes a special type of loop). Note that the primitive function `length`() gives as output the number of elements of the vector passed in input. Examples 14-16 complete this section by showing three typical algorithms for vectors.

Example 14: Find a number into a vector

```
N := Numeric
pos := Numeric
x := Numeric[N]
y := Numeric

Begin
  x[] = concatenate(...)
  pos = find_vector(x[],y)
  print("The number has position" pos)
End

find_vector := function(a[],b ){
found := Numeric
N := Numeric
N = length(a[])
for(i in 1:N ){
if(a[i]==b){found = i}
}
return(found)
}
```

Example 15: Find maximum and its position

```
N,maximum, pos_maximum := Numeric
x := Numeric[N]
res := Numeric[2]

Begin
  x[] = concatenate(...)
  res[] = find_maximum(x[])
  print("The maximum is" res[1])
  print("The maximum has position" res[2])
End

find_maximum := function(a[] ){
N,found_max,found_pos := Numeric
out := Numeric[2]
N = length(a[])
found_max = a[1]
pos_max = 1
for(i in 2:N ){
if(a[i] > found){
found_max = a[i]
found_pos = i }
}
out[1] = found_max
out[2] = pos_max
return(out)
}
```

Example 16: Sorting a vector (bubble sort)

```
N := NUMERIC
x,y := NUMERIC[N]

BEGIN
  x[] = concatenate(...)
  y[] = sort(x[])
  print("The sorted vector is" y[])
END

sort := function(a[] ){
max,c,N,K := NUMERIC
next := BOOLEAN
N = length(a[])
next = 1
K = N
while(next == 1 ){
max = K
next = 0
for(i in 1:(max-1) ){
if(a[i] > a[i+1]){
c = a[i]
a[i] = a[i+1]
a[i+1] = c
next = 1
K = i }
}
}
return(a)
}
```

7.2 Matrices

A **matrix** is a set of parallel vectors concatenated column-wise or row-wise. They are two-dimensional arrays declared as follows:

```
(name) := NUMERIC[N,M]
(name) := BOOLEAN[N,M]
(name) := STRING[N,M]
```

where N indicates the number of rows whereas M the number of columns. As for vectors, $N > 0$ and $M > 0$. A matrix with $N = M = 1$ is a singleton, a matrix with $M = 1$ and $N > 0$ is a column-wise vector, whereas a matrix with $M > 0$ and $N = 1$ is a row-wise vector. Matrices can be manipulated by means of linear algebra: addition, subtraction, multiplication, and inversion are allowed if matrices have the correct dimensions. Similarly to vectors, matrices can be populated using a special for-loop (i.e., two nested for-loop) or via `matrix(data,nrow,ncol,by)`, a primitive function used to populate matrices. It works as shown below:

```
A := NUMERIC[2,3]
A[] = matrix(data=concatenate(1,2,3,3,2,1),nrow=2,ncol=3,by=ROWS)
```

where `by` is the argument controlling how the population has to be made: row-wise (`ROWS`) or column-wise (`COLS`). When `by=ROWS` the resulting matrix is:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$$

By contrast, when `by=COLS` the output is:

$$A = \begin{bmatrix} 1 & 3 & 2 \\ 2 & 3 & 1 \end{bmatrix}$$

Two important primitive functions working with matrices are `NROW(X[])` and `NCOL(X[])` computing the number of rows and columns of a matrix. Examples 17-18 show typical programs involving matrices.

statement	a[]	max	c	N	K	next	i	true/false	output
function call	[2 1 7 4]								
length(a[])				4					
next = 1						1			
K = N					4				
next == 1								V	
max = K		4							
next = 0						0			
for-loop							1		
a[i] > a[i+1]								V	
c = a[i]			2						
a[i] = a[i+1]	[1 1 7 4]								
a[i+1] = c	[1 2 7 4]								
next = 1						1			
K = i					1				
for-loop							2		
a[i] > a[i+1]								F	
for-loop							3		
a[i] > a[i+1]								V	
c = a[i]			7						
a[i] = a[i+1]	[1 1 4 4]								
a[i+1] = c	[1 2 4 7]								
next = 1						1			
K = i					3				
next == 1								V	
max = K		3							
next = 0						0			
for-loop							1		
a[i] > a[i+1]								F	
for-loop							2		
a[i] > a[i+1]								F	
next == 1								F	
return()									[1 2 4 7]

Table 6: Table commands × variables for the function sort() in Example 16

Example 17: Compute the trace of a square matrix (sum of the elements of the diagonal)

```
I,J := Numeric
X := Numeric[I,J]
sum := Numeric
Begin
  sum = 0
  X[]  = matrix(data=concatenate(...),nrow=I,ncol=J,by=ROWS)
  if(I != J){print("The matrix is not a square matrix")}
  else{
  for(i in 1:I ){ for(j in 1:J ){ if(i==j){sum = sum+X[i,j]} } }
  }
  print("The trace of the matrix is" sum)
End
```

Example 18: Given a square matrix, find the maximum element of the diagonal

```
I,J := Numeric
X := Numeric[I,J]
max := Numeric
Begin
  sum = 0
  X[]  = matrix(data=concatenate(...),nrow=I,ncol=J,by=ROWS)
  if(I != J){print("The matrix is not a square matrix")}
  else{
  max = find_maximum_diagonal(X[])
  }
  print("The maximum of the diagonal is" max)
End

find_maximum_diagonal := function(X[] ){
N,max := Numeric
x := Numeric[N]
x  = diag(X[])
max  = find_maximum(x)
return(max)
}

diag := function(X[] ){
N := Numeric
x := Numeric[N]
N = NROW(X[])
for(i in 1:N ){ if(i==j){x[i] = X[i,j]} }
return(x[])
}

find_maximum := function(a[] ){
N,found_max := Numeric
N = length(a[])
found_max = a[1]
for(i in 2:N ){
if(a[i] > found_max){ found_max  = a[i] }
}
return(found_max)
}
```

8 Lists

Whilst arrays are structured data containing information of the same type, **lists** (or **records**) are particular structures allowing for different type of information. In this way, programs can store - in the same structure - numbers and strings, as well as variables, vectors, and matrices. An example of list declaration is as follows:

```
(name) := List[3]{x := Numeric, y := String, A := Numeric[2,3]}
```

which indicates a list of three elements containing two single variables (numeric and boolean) and a matrix. Since lists are composite structures, they are populated by populating their single components. For example:

```
z := List[2]{x := Numeric, A := Numeric[2,2]}
z[[1]]  = 3.5
z[[2]] = matrix(data=concatenate(2,3,2,5),nrow=2,ncol=2,by=ROWS)
```

Unlike for vectors and matrices, programs access the elements of a list by means of double-square brackets. Examples 19-20 shows two programs using lists.

Example 19: Register users' personal information

```
N,x := Numeric
users := List[4]{first_name := String, last_name := String, place_birth := String,
year_birth := Date, occupation := String }

Begin
  N = read("Insert the number of users to register")
  for(i in 1:N ){ # populate the list
  users[i][[1]]  = read("Insert first name of user number" i)
  users[i][[2]]  = read("Insert last name of user number" i)
  users[i][[3]]  = read("Insert place of birth of user number" i)
  users[i][[4]]  = read("Insert year of birth of user number" i)
  users[i][[5]]  = read("Insert occupation of user number" i)
  }
  x = read("Insert user's number)
  print("First name" users[i][[1]])
  print("Last name" users[i][[2]])
  print("Place of birth" users[i][[3]])
  print("Year of birth" users[i][[4]])
  print("Occupation" users[i][[5]])
End
```

Example 20: Given a square matrix, find diagonals and maxima

```
I,J := Numeric
X := Numeric[I,J]
results := List[4]{diag1 := Numeric[I], diag2 := Numeric[I], max1 := Numeric, max2 := Numeric}

Begin
   sum = 0
   X[]  = matrix(data=concatenate(...),nrow=I,ncol=J,by=ROWS)
   if(I != J){print("The matrix is not a square matrix")}
   else{
   results = find_diagonals_maxima(X[])
   }
   print("The main diagonal is" results[[1]])
   print("The secondary diagonal is" results[[2]])
   print("The maximum of the first diagonal is" results[[3]])
   print("The maximum of the second diagonal is" results[[4]])
End

find_diagonals_maxima := function(X[] ){
N,max := Numeric
x := Numeric[N]
out := List[4]{diag1 := Numeric[I], diag2 := Numeric[I], max1 := Numeric, max2 := Numeric}
x  = diag1(X[])
max  = find_maximum(x)
out[[1]] = x
out[[3]] = max
x  = diag2(X[])
max  = find_maximum(x)
out[[2]] = x
out[[4]] = max
return(out)
}

diag1 := function(X[] ){
N := Numeric
x := Numeric[N]
N = NROW(X[])
for(i in 1:N ){ if(i==j){x[i] = X[i,j]} }
return(x[])
}

diag2 := function(X[] ){
N := Numeric
x := Numeric[N]
N = NROW(X[])
for(i in 1:N ){ for(j in N:1 ){ x[i]  = X[i,j] } }
return(x[])
}

find_maximum := function(a[] ){
N,found_max := Numeric
N = length(a[])
found_max = a[1]
for(i in 2:N ){
if(a[i] > found_max){ found_max  = a[i] }
}
return(found_max)
}
```

9 Correspondence between ARLA and R commands

ARLA	R	notes
Declaring variables `(name variable)` := TYPE	No formal declaration is required	Variables are automatically declared during their use
Assignment `x` = 2.5, `x` = "hello"	`x` = 2.5, `x` ← "hello"	Both symbols " = " and "←" can be used to assign values to variables/objects
Basic statements **read**(...), **print**(...)	`readline(...)` or other data-input functions	In R, values can be read and stored in different ways, e.g. using `read.csv(...)` for external csv file or using the console directly
Selection **if**`(...)`{...}, **else**{...}	`if(...){ ... }` `else{...}`	
Logical operations ==, ! =, >=, >, <=, <	==, ! =, >=, >, <=, <	
Loops **for**(`index in index set`){...} **while**(`condition`){...}	`for(index in index set){...}` `while(condition){...}`	
Functions `name_function` := **function**(`input`){...} `x` = name_function(input)	`name_function = function(input){...}` `x` = name_function(input)	
Declaring vectors `name` := NUMERIC[N]	No formal declaration is required	Vectors are automatically declared during their use
Populating vectors `x[]` = `concatenate(...)`	`x` = `c(...)`	
Declaring matrices `name` := NUMERIC[N,M]	No formal declaration is required	Matrices are automatically declared during their use
Populating matrices `A[]` = `matrix(data=concatenate(..), nrow=N,ncol=M,by=ROWS)`	`A = matrix(data=c(...),nrow=N,ncol=M, byrow=TRUE)`	
Declaring lists `name_list` := **List**[N]{ `name_element` := TYPE,...}	No formal declaration is required	Lists are automatically declared during their use
Populating lists The same as for variables and arrays	The same as for variables and arrays	Populating elements of a list is the as populating variables and arrays
Access elements of arrays and lists `x[i]`, `A[i,j]`, `z[[k]]`	`x[i]`, `A[i,j]`, `z[[k]]`	

Table 7: Correspondence between ARLA and R languages

10 A complete example with ARLA and R

There are many versions of the algorithm to compute the square root of a positive number S (e.g., see `https://en.wikipedia.org/wiki/Methods_of_computing_square_roots`). In this section, we will describe an algorithm using the famous *Heron's method*, which computes the square root of S via the approximation:

$$\sqrt{S} = \lim_{n\to\infty} x_n$$

where the n-th element of the succession is defined as:

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{S}{x_{n-1}}\right)$$

by ensuring that $x_0 > 0$ (starting condition).

Example 20: Heron's method with ARLA

```
N := Numeric
x := Numeric[N]
S := Numeric

Begin
  S = read("Insert a positive number")
  N = read("Insert the number of steps to compute the square root")
  x[1] = 1.5 # starting value
  if(S>0){
  for(i in 2:N ){ x[i]  = 0.5(x[i-1] + S/x[i-1]) }
  }
  else{print(S "must be greater than zero")}
  print("The square root of" S "is" x[N])
End
```

Example 20: Heron's method translated in R

```
N = 20 # it can be changed
x = rep(NA,N) # populating the vector x with NAs
S = 30 # it can be changed
x[1] = 1 # it can be changed

  if(S>0){
  for(i in 2:N ){ x[i]  = 0.5(x[i-1] + S/x[i-1]) }
  }
  else{print(paste(S,"must be greater than zero", sep=" "))}
  print(paste("The square root of", S, "is", x[N], sep=" "))
End
```

statement	x[]	S	N	i	true/false	output
read("Insert a positive number")		8				
read("Insert the number of steps...")			5			
x[1] = 2.5	[2.5 ...]					
S>0					V	
x[2] = 0.5(x[1] + S/x[1])	[2.5 2.85 ...]			2		
x[3] = 0.5(x[2] + S/x[2])	[2.5 2.85 2.8285 ...]			3		
x[4] = 0.5(x[3] + S/x[3])	[2.5 2.85 2.8285 2.8284 ...]			4		
x[5] = 0.5(x[4] + S/x[4])	[2.5 2.85 2.8285 2.8284 2.8284]			5		
print("The square root "...)						2.8284

Table 8: Table commands × variables for the Heron's algorithm

Basics of programming language: An introduction using ARLA

Exercises

Antonio Calcagnì
DPSS - University of Padova

Contact: antonio.calcagni@unipd.it
Version: 1.0 (October, 2019)

Exercise 1. Write an algorithm that reads three real numbers x, y, z and compute the maximum among them.

Exercise 2. Write an algorithm that reads four real numbers x, y, q, z, compute (i) maximum $a_{\max}$ and minimum $a_{\min}$ among them, (ii) the difference $a = a_{\max} - a_{\min}$, and verifies whether $a > 0$.

Exercise 3. Write an algorithm that reads three real numbers x, y, z and computes the value of the following expression:

$$w = \frac{\min(x, y)}{z^2} + \max(x, z)$$

TIPS: min() and max() are not primitive functions in this exercise and need to be written by scratch.

Exercise 4. Write an algorithm that reads two real numbers x and y and execute the following instructions:

1. if $\lfloor \min(x, y) \rfloor$ is an even number then evaluates the expression $z = x^{|y|}$
2. if $\lfloor \min(x, y) \rfloor$ is an odd number then evaluates the expression $z = x^{|y-x|}$
3. outputs the value of z

TIPS: The value of $\lfloor a \rfloor$ can be computed using the primitive function `floor()`, the absolute value $|a|$ can also be computed by means of the primitive `abs()`, whereas to verify whether a number is odd or even, one can use the primitive `mod(a,2)` which gives as output `True` if the number is even.

Exercise 5. Write an algorithm that reads three integers x, y, z and evaluates the following instructions:

1. computes $q = x + y + z$ and, if $q < 0$, transforms the input such that q is always greater than zero
2. if $q < \min(a, b, c)$ then computes $z = s^2 + \frac{1}{2}s^3$
3. if $q > \min(a, b, c)$ then computes $z = \frac{s^2}{2} + \frac{1}{s^3}$

4. outputs z

Exercise 6. Write an algorithm that reads two real numbers x and y and computes the following expression:

$$z = \max(|x|, |y|) - \min(x, y) + (x - y)^2$$

If $z < 0$ then asks for a third number q and evaluates whether $z < q$.

Exercise 7. Write an algorithm that reads a real number a, determines its sign $s = \text{sign}(a)$, and outputs a boolean value $\{1, 0\}$ indicating its sign s.

Exercise 8. Write an algorithm that reads N numbers, computes their sum s until the condition $s > q$ is met.
TIPS: q is a constant stored before the execution of the sum-loop.

Exercise 9. Write an algorithms that reads two intervals $i_1 = [a, b]$ and $i_2 = [c, d]$ and evaluates the following instructions:

- if $i_1 \subset i_2$ then computes $h = (\max(a, b) - \min(a, b)) \frac{1}{j}$, where $j = \frac{a+b}{2}$
- if $i_1 \not\subset i_2$ then computes $h = (\max(a, b) - \min(a, b)) \frac{1}{j}$, where $j = \frac{a}{|b|+\epsilon}$ and $\epsilon = \max(a, b, c, d)^{\frac{1}{3}}$
- if $i_1 = i_2$ then computes $h = (\max(a, b) - \min(a, b))$
- outputs h

Exercise 10. Write an algorithm that reads two points on $\mathbb{R}^2$, i.e. $\mathbf{x} = (x_1, y_1)$ and $\mathbf{y} = (x_2, y_2)$ and determines if the line passing through these points also passes through the origin $\mathbf{c} = (0, 0)$ or not.
TIPS: To check for the line passing through the origin, the equation $x_1(y_2 - y_1) = y_1(x_2 - x_1)$ must be satisfied.

Exercise 11. Write an algorithm that reads a $I \times 1$ vector of real numbers $\mathbf{x} = (x_1, \ldots, x_i, \ldots, x_I)$ and gives as output the vector $\mathbf{y} = (x_I, \ldots, x_i, \ldots, x_1)$.

Exercise 12. Write an algorithm that reads two natural numbers n_a and n_b and determines the vector $\mathbf{x}$ containing the sequence of numbers included between n_a and n_b.

Exercise 13. Write an algorithm that reads the array $\mathbf{x}_{n\times 1}$ of real numbers and evaluates the following expressions:

1. $x^* = \sum_{i=1}^{n} x_i$
2. $\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$
3. $x^\dagger = x^* / (\max(\mathbf{x}) - \min(\mathbf{x}))$
4. $z = (x^* - 2x^\dagger)$
5. write as output z

Exercise 14. Write an algorithm that reads a vector of integers $\mathbf{n}_{I\times 1}$ and computes the vector $\mathbf{f}_{I\times 1}$ containing the frequencies of the elements of $\mathbf{n}_{I\times 1}$.
TIPS: Write a function that counts how many times the elements of $\mathbf{n}$ occur.

Exercise 15. (*cont. Exercise 14*) Write an algorithm that reads as input the vector of frequencies $\mathbf{f}_{I\times 1}$ and computes the vector $\mathbf{s}_{I\times 1}$ where the generic element s_i is defined as follows:

$$s_i = s_{i-1} + f_i$$

Note that $s_1 = f_1$ by definition.

Exercise 16. Write an algorithms that reads two vectors of reals $\mathbf{x}_{n\times 1}$ and $\mathbf{s}_{n\times 1}$ and computes the following expression:

$$v_i = x_i + (x_{i+1} - x_i)\frac{0.5 - s_i}{s_{i+1} - s_i}$$

TIPS: The vector $\mathbf{s}$ contains the cumulative frequencies of $\mathbf{x}$.

Exercise 17. Write an algorithm to approximate the computation of following expression:

$$\int_1^N z^2 + \sin(z)dz \;\approx\; \frac{N-1}{N}\sum_{i=1}^{N} z_i^2 + \sin(z_i)$$

given a vector of reals $\mathbf{z}_{N\times 1}$ as input.

Exercise 18. Write an algorithm that compute the square root of a number $s > 0$ using the Heron's approximation:

$$\sqrt{s} = \lim_{n\to\infty} x_n$$

where the generic element x_n is defined as:

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{s}{x_{n-1}}\right)$$

with $x_1 = 1.5$. The iterative approximation must be executed until the condition $(x_n - x_{n-1}) < \epsilon$ is met, with ϵ being a user's defined small real number (e.g., $\epsilon = 0.001$).

Exercise 19. Write an algorithm that reads two vectors $\mathbf{x}_{n\times 1}$ and $\mathbf{y}_{n\times 1}$ containing pairs of coordinates (integers), and computes the Manhattan distance between all pairs of points stored in the input vectors:

$$d = \sum_{i=1}^{n}\sum_{j=i+1}^{n} |x_i - x_j| + |y_i - y_j|$$

TIPS: Use a brute-force approach.

Exercise 20. Write an algorithm that reads a vector of strings $\mathbf{t}_{n\times 1}$ and a single string s, and verifies if s is contained in $\mathbf{t}_{n\times 1}$.

Exercise 21. Write an algorithm that reads a $N \times M$ matrix $\mathbf{X}$ with $N = M$ and computes the following quantities:

1. $q = \sum_{i=1}^{N} \text{diag}(\mathbf{X})_i + \max(\mathbf{X})$
2. $p = \prod_{i=1}^{N} \text{diag}(\mathbf{X})_i + \min(\mathbf{X})$
3. if $q > p$ outputs p, otherwise outputs q

Exercise 22. Write an algorithm that reads a $N \times M$ matrix $\mathbf{X}$ with $N = M$, transforms the input array into:

$$\mathbf{x} = (X_{1,1}, \ldots, X_{1,M}, \ldots, X_{i,M}, \ldots, X_{N,M})$$

and outputs $\mathbf{x}$.

Exercise 23. Write an algorithm that reads a $I \times J$ matrix of reals $\mathbf{A}$ and executes the following instructions:

1. computes the sum of the elements below the main diagonal (excluding the elements along the diagonal)

2. computes the maximum of the elements upper the main diagonal (excluding the elements along the diagonal)
3. computes the minimum of the elements along the main diagonal
4. outputs all the results

Exercise 24. Write an algorithm that reads a matrix of reals $\mathbf{X}_{I\times J}$ and computes the following quantities:

1. $a = \sum_{i=1}^{I} \sum_{j=1}^{J} \mathbf{X}_{i,j}$ with $i \neq j$
2. $b = \sum_{l\in\mathcal{L}} \sum_{h\in\mathcal{H}} \mathbf{X}_{l,h}$ with $\mathcal{L}$ being the set of *even* indices and $\mathcal{H}$ the set of *odd* indices
3. computes $q = a + b$ if $a > b$, otherwise $q = \frac{1}{2}(a - b)^2$
4. outputs q

Exercise 25. Write an algorithm that reads two numbers $N > 0$, K and finds a square matrix $\mathbf{A}_{N\times N}$ such that the sum of elements in every row and column is equal to K.

Exercise 26. Write an algorithm that reads two numbers $N > 0$, K and finds a square matrix $\mathbf{A}_{N\times N}$ such that

$$\sum_i \text{diag}(\mathbf{A})_i = K\,(\max(\mathbf{A}) - \min(\mathbf{A}))$$

Exercise 27. Write an algorithm that reads a positive integer I and two vectors of *different* integers $\mathbf{n}_{I\times 1}$, $\mathbf{m}_{I\times 1}$, and finds a matrix $\mathbf{A}_{I\times I}$ such that

$$a_{i,j} = n_i + m_j$$

for all $i, j \in \{1, \ldots, I\}$.

Exercise 28. Write an algorithm that reads a vector of characters $\mathbf{s}_{K\times 1}$ and determines the matrix $\mathbf{S}_{N\times M}$ filled by all the characters containing in $\mathbf{s}$, where $N = \lfloor\sqrt{K}\rfloor$ and $M = \lceil\sqrt{K}\rceil$.
TIPS: The values of $\lfloor a\rfloor$ and $\lceil a\rceil$ can be computed using the primitive function `floor()` and `ceil()`, respectively.

Exercise 29. Write an algorithm that reads a matrix of reals $\mathbf{X}_{N\times N}$ and an integer N, and sorts $\text{diag}(\mathbf{X})$ in descending order.

Exercise 30. Write an algorithm that reads two matrices of the same order $\mathbf{A}_{N\times N}$, $\mathbf{B}_{N\times N}$, and defines a new matrix $\mathbf{C}_{N\times N}$ containing only corresponding common elements and place `NA` at the positions where elements are different.